DESIGN

EMBEDDED SYSTEMS **PART 4: FUTURE DIRECTIONS**

Embedded designers face tough choices

BY BERNARD C. COLE

Even as embedded systems designers and software developers cope with a continuing proliferation and expansion of existing architectures, they also face radically different control alternatives: digital signal processing, fuzzy logic and neural networks.

"What embedded designers are faced with," said Emdad Kahn, director of intelligent systems for the Embedded Systems Division at National Semiconductor (Santa Clara, Calif.), "is a choice between a methodology, DSP, which is familiar in that it deals with the implementation of algorithms, but that is extraordinarily complex, and fuzzy and neural nets, which at heart result in very simple and concise code, but require a leap of faith to a rules-based methodology."

The familiarity of a common algorithmic approach to embedded code implementation, combined with the fact that the methodology has received extensive support from both chip vendors and software developers over the past five years, has led to widespread acceptance of DSP, said Christopher Hale, applications engineer at Motorola Inc.'s Advanced Microcontroller Division (Austin, Texas).

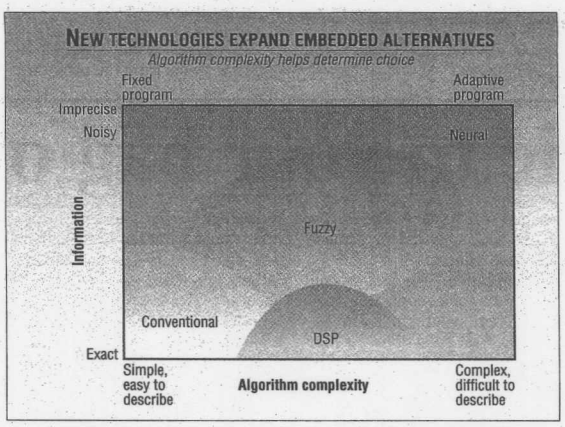
DSP functions are being integrated onto the same chip as traditional CPUs. This ranges from DSP-specific chips, such as the eight-bit PIC17CXX from Microchip Technology (Chandler, Ariz.), at the low end, to 32-bit implementations, such as TI's TMS32c30/40, at the high end. At the low end of the embedded market, DSP functions may range from the addition of a few simple instructions to incorporation of multiply accumulator units, such as in Motorola's 16-bit 68HC16, to full-fledged floating-point units and DSP-based co-processors in high-end processors such as Intel's i960 and Motorola's CPU32 family of application-specific 32-bit processors.

In the view of Ken Robinson, engineering manager at Bally Gaming Corp. (Las Vegas, Nev.), whose company uses Z80s, 68000s and 68332s, such integrated solutions have made the transition to DSP much easier. "The fact that the features and instructions are there on chip makes it much easier to experiment with such functions when they seem needed," he said.

But that ease and the familiarity of the DSP's algorithmic approach can be deceptive, said Mark Clayton, applications engineer at Ariel Corp. (Highland Park, N.J.), a DSP board and tool vendor: "In the development process, there are conditions that in traditional architectures would indicate a bug in your code, which in DSP are normal operating procedure. And it takes some experience with DSP to know the difference."

A similar and even steeper learning curve is in store for engineers who want to become proficient in such new techniques as neural networks and fuzzy logic. At the silicon level, a range of ICs is becoming available, from vendors ranging from such Japanese firms as NEC to U.S. companies such as Intel and startups such as Togai Infralogic Corp. (Irvine, Calif.) and American Neurologix Inc. (Sanford, Fla.).

Despite the chips' increasing availability cost will rule out their use unless they are perceived as the only solution to the problem, said David Brubaker, president of the Huntington Group (Menlo Park, Calif.), an embedded design consulting firm. "Other than such early adopters," he said, "fuzzy logic and neural networks will follow the pattern that has occurred with DSP: first implementation in software on traditional architectures, then with some assists in the instruction set, then with modifications to the general-purpose architecture to accommodate the methodologies and then hybrid architectures."



In the fuzzy-logic segment of the market, there is growing acceptance among semiconductor vendors as well as fuzzy logic vendors such as Aptronix Inc. (San Jose, Calif.), Togai Infotronix and Omron Ltd. (Tokyo) that the best way to penetrate the mainstream of the embedded market is with an evolutionary approach. The idea is to entice designers to this new methodology with tools that allow them to implement fuzzy designs with existing microcontrollers and processors.

"Properly designed and implemented fuzzy logic can make traditional embedded designs more code-efficient and faster, even without specialized silicon," said Brubaker, "but that will not be enough for it to make it an accepted methodology for mainstream designers. " And while neural networks are much more dependent on available silicon, easy-to-use tools are extremely important in this area as well, he said, if the market is to grow.

A variety of tools is becoming available, said Brubaker, but as with all first-generation efforts, there are pros and cons. In the fuzzy logic area, Togai Infralogic provides the FC110 fuzzy logic hardware and associated development system. Omron has a similar solution, except that its FP3000 chip is actually a dedicated peripheral rather than a microprocessor.

Aptronix offers a fuzzy development system, the FIDE, that is PC-based, and once developed can be automatically converted to assembly code for a variety of processor targets, including Motorola's 68HC11, 6805 or 68HC05. Inform Technology (Achem, Germany) offers a FuzzyTech package that currently supports Intel's MCS96 family and Siemens's 80C166 microcontrollers, among others.

A processor-independent package from Byte Craft Ltd. (Ontario, Canada), called Fuzz-C, is a DOS-based tool that accepts fuzzy rules as text input and generates a C-language source file that can then be compiled with the rest of an application's code.

On the neural network side, a number of fairly advanced development tools have recently become available. NeuroDimension Inc. (Gainsville, Fla.), is offering Neurosolutions, which uses Objective C for the user interface and generates C++ code. Gensym Corp. (Cambridge, Mass.), offers its G2 Realtime Expert System and NuronLine, a visual object-oriented tool for building neural networks. Cynex & HTC (Wells, Australia), is offering NetTools, an integrated set of programs for simulating neural networks and other logic circuits, including fuzzy logic.

For consulting engineer Brubaker, the drawbacks to such packages lie not in the products themselves, but in the fact that for both neural nets and fuzzy logic there is still no generally accepted formal design methodology. "It is still pretty much seat of the pants as far as I can see," he said, "with no good metrics. As a result there is not only no good way to determine when to use such techniques, but also how better they will be than traditional methods."

For fuzzy tools in particular, he said, there is still no way to determine the quality of the code. "The debug capabilities of such packages, when they do exist at all, are still relatively unsophisticated, compared to what embedded designers have become accustomed to."

Continued on page 66

INTRODUCTION

The future has arrived for designers of embedded systems, and it is not a simple one. Designers face a bewildering array, ranging from application-specific CPUs and digital signal processors to fuzzy logic and neural nets.

The choices offer opportunities, but bring confusion as well. The promise of all this technology is tempered by the task of sorting through it, said Gil Porter, a technology fellow at Delco Electronics Corp. (Kokomo, Ind.). "We are kept more than busy enough just coping with the proliferation of existing architectures," Porter said, "with the higher speeds, the more complex emulation and debug challenges, and with new alternatives such as RISC, which are proliferating downward." As for the up-and-coming contenders, he said, "it's hard to start thinking about new alternatives such as DSP and fuzzy logic, unless there is a clear and present need—unless that is the only way to solve the problem at hand. "

In this last part of our Embedded Systems series, we attempt to meet the designers' concerns by exploring the future of embedded design while staying pegged to the practical tools of today. —*B.C.*

Embedded Systems: Part 4

Clear thinking on fuzzy linguistics

By Walter Banks
President
Byte Craft Ltd.
Waterloo, Ontario

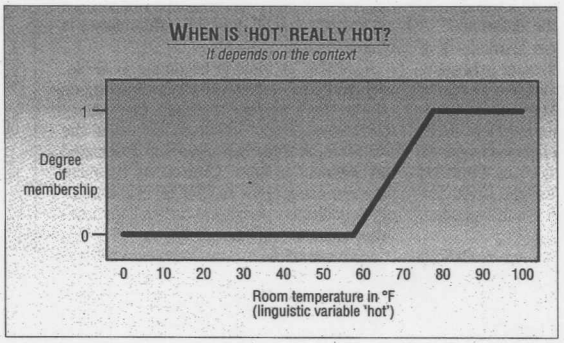
I've watched the effort to find uses for fuzzy logic—an almost 30-year-old "new" technology—from a front-row seat. As I began to consider the technology in terms of how the growing number of fuzzy-logic success stories might be implemented, I turned to language theory to determine why linguistic variables are important in describing and solving problems on computers.

Linguistic variables are central to fuzzy-logic manipulations. Linguistic variables hold values that are uniformly distributed between 0 and 1, depending on the relevance of a context-dependent linguistic term. For example, we can say the room is hot or the furnace is hot. The linguistic variable "hot" differs in meaning depending on whether we apply it to the room or to the furnace.

A linguistic variable with an assigned value of 0 is not true; an assigned value of 1 indicates that the term is true. The "linguistic variables" used in everyday speech convey a surprising amount of relative information about our environment or an object under observation.

The relationship between crisp numbers and linguistic variables is generally well understood. The computer require that there be a formal way of describing the linguistic variable in crisp terms that the computer can handle.

The figure shows the relationship between measured room temperature and the linguistic



linguistic variable "hot" in the accompanying figure has a value between 0 and 1 (where 0 is not hot at all and 1 is undeniably hot) over the crisp range of 60 to 80 degrees F. For each crisp number in a variable space (such as "room"), a number of linguistic terms may apply. Linguistic variables in a term "hot." In the space between "hot" and "not hot," the temperature is, to some degree, both. The horizontal axis shows the measured or crisp value of temperature. The vertical axis describes the degree to which a linguistic variable fits with the crisp, measured data.

Most fuzzy-logic support software has a form similar to the declaration shown in the figure. In this case, a crisp variable ("room") is associated with a linguistic variable ("hot"), which is defined using four breakpoints from the graph.

We often use linguistic references enhanced with crisp definitions. Consider these instructions, quoted from a package of minestrone soup mix, that show just how common linguistic references are in our descriptive language: "Empty contents into a saucepan; add 4½ cups (1 liter) cold water."

The soup instructions are in both the crisp and fuzzy domains. The linguistic variable "saucepan," for example, is qualified by the quantity of liquid that is expected. One liter is not exactly 4½ cups, but the measurement is accurate enough (within 6.5 percent) for the job at hand. "Cold water" is a linguistic variable that describes water whose temperature is between the freezing point (at which water is unequivocally cold) and some higher temperature at which the water is cold to some degree.

The power of any computer language stems from the ability to describe a given problem in terms that are relevant to that problem. Linguistic variables are relevant for many applications that involve a human interface. The fuzzy-logic success stories involve implementations of tasks commonly performed by humans but not easily described in crisp terms. Rice cookers, toasters, washing machines, environment control, subway trains, elevators, camera focusing and picture stabilization are just a few examples. Linguistic variables do not simplify an application or its implementation, but they do provide a convenient tool to describe a problem.

Applications may be computed in either the fuzzy, linguistic domain or the conventional, crisp domain. Non-linear problems, such as process control in an environment that varies considerably from usage to usage, yield very workable results with impressively little development time when solved using fuzzy logic. Though fuzzy logic is not essential to solving non-linear control problems, it helps in describing some of the possible solutions.

Lotfi Zadeh, the originator of

Continued on page 49

Adding fuzzy logic to your bag of tricks

By James M. Sibigtroth
Member, Technical Staff
Motorola Inc.
Microcontroller Technical Group
Austin, Texas

While fuzzy logic introduces some new tools and methodology to embedded system development, it also extends the range of application problems that can be solved using inexpensive microcontrollers. Fuzzy logic makes it possible to capture the knowledge of application experts without requiring them to learn microcontroller programming.

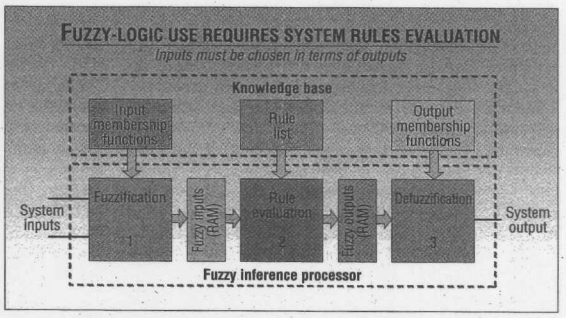
In a short time, fuzzy logic will become recognized as a basic programming technique and a required tool in every embedded programmer's bag of tricks. Much of the work can be done using familiar MCU development tools, but new fuzzy logic development software is very helpful in some parts of the design and debug processes.

A quick review of fuzzy logic starts with an understanding of control applications, which involve an input to output transformation. At any given time, the system produces an appropriate output based on the current values of system inputs. Depending on the application, this input-to-output relationship may be expressed as a mathematical formula or some Boolean logical function. Fuzzy logic is just a new way to define this relationship which happens to work very well when the input-to-output relationship is not easily expressed using traditional digital techniques.

The accompanying figure shows a block diagram of a fuzzy logic software system. System inputs enter at the left and system outputs exit at the right. The upper half of the figure is called the knowledge base, and all application specific information is contained in this block. The lower half of the figure is called the fuzzy inference engine. The inference engine can be used, without changes, for many different applications.

Each of the three parts of the inference engine has a corresponding data structure in the knowledge base. RAM data structures connect one step of the inference process to the next. The software in the inference engine can be developed using traditional MCU development tools and techniques. The information in the knowledge base is more easily managed with a fuzzy logic development tool such as the knowledge base generator.

"Fuzzification," the first step in fuzzy processing, compares current system inputs to membership function definitions in the knowledge base to produce fuzzy inputs in a RAM data structure.



Each fuzzy input is interpreted as the degree to which the corresponding linguistic label is true. The value $00 corresponds to false and $FF corresponds to true. You could also interpret the fuzzy input value as a binary fraction between 0.0 and 0.996.

The rule evaluation step processes a list of rules from the knowledge base. Current fuzzy input values from the fuzzification step are used to produce fuzzy outputs in a RAM data structure. The most common rule evaluation operators are the mathematical minimum for the AND operator and the mathematical maximum for the OR operator. There has been a lot of research into the use of other operators for special applications.

A rule such as "If Temperature is Hot and Pressure is High then Blower is High", can be reduced to a list of pointers which can be stored in ROM as part of the knowledge base. The linguistic expressions "Temperature is Hot" and "Pressure is High" are antecedents and each is directly related to one fuzzy input. The linguistic expression "Blower is High" refers to a fuzzy output. Min-max rule evaluation simply looks up each rule antecedent, saving the smallest one (min) in an accumulator.

This accumulator value is interpreted as the truth value for the whole rule. This truth value for the rule is then stored to each consequent fuzzy output, unless there is already a larger value in the fuzzy output (max).

In a software fuzzy inference engine, rules are processed sequentially but they could be processed in parallel in a hardware system dedicated to solving fuzzy problems. The obvious benefit would be speed. The cost is dedicated logic and lack of flexibility to make changes in the rule evaluation strategy.

The final "defuzzification" step in the fuzzy inference engine combines the fuzzy outputs from the rule evaluation step into a single composite system output. This is comparable to the way people make control decisions. We consider several rules with different proposed actions and associated

Continued on page 54

Embedded Systems: Part 4

Call for linguistics approach

Continued from page 46

fuzzy logic, noted that ordinary language contains many descriptive terms whose relevance is context-specific. I can, for example, say, "The day is hot." That statement conveys similar information to most people. Indeed, simply saying "It's hot today" may convey the information more effectively than saying, "The temperature is 35°," since the listener, depending on whether he's accustomed to temperature readings in °C or °F, could infer the latter phrase to mean either that conditions are very warm or that they're very cool.

"The day is muggy" implies two pieces of information: The day is hot, and the relative humidity is high. A day can be hot or muggy or cold or clammy. In common usage, linguistic variables are often overlapping. "Muggy" implies both high humidity and hot temperatures. The variable "day" may have an extensive list of linguistic variables computed in the fuzzy domain associated with it (muggy, humid, hot, cold, clammy). If "day" is a linguistic variable, it doesn't have a crisp number associated with it. Thus, although we can say the day is hot or muggy, assigning a value to "day" would be meaningless. All of the linguistic members associated with "day" are based on fuzzy-logic equations.

New variable type

When fuzzy logic is used in an application program, it adds linguistic variables as a new variable type. We might implement an air-conditioner controller with a single fuzzy statement: "IF room is hot THEN air conditio000189 on." We can extend basic air-conditioner control to behave differently depending on the different types of day.

The math developed to support linguistic variable manipulation conveniently implements an easy method to switch smoothly from one possible solution to another. Thus, unlike a conventional control system, which implements a single, well-behaved control function, a fuzzy-logic control system design can apply many solutions (or rules) to a single problem, and the combined solutions can be appropriately weighted to the controlling action.

Computers, especially those in embedded applications, can be programmed to perform calculations in the fuzzy domain rather than the crisp domain. Fuzzy-logic manipulations take advantage of the fact that linguistic variables are only resolved to crisp values at the resolution of the problem—a kind of self-scaling feature that is objective-driven rather than data-driven.

If a room is to be kept comfortable, the temperature and humidity need to be kept only within the fuzzy comfort zone. Any calculations that have greater accuracy than the desired result are redundant and require more computing power than is needed.

The next great leap in computing power may actually come from problem organization rather than processor power and speed. Fuzzy logic is not the only way to achieve reductions in computing requirements, but it is the best of the methods suggested so far to achieve that goal.

Linguistic variables are taking their place alongside such other data types as character, string, real and float. They are, in some ways, an extension of the already familiar enumerated data types that are common in many high-level languages. In my view, the linguistic domain is just another of the tools that application developers have at their disposal to communicate clearly. When applied appropriately, fuzzy-logic solutions are competitive with conventional implementation techniques—with considerably less implementation effort.

Embedded Systems: Part 4

Nothing fuzzy about processor choice

By Joe Altnether
Fuzzy Logic Program Director
Intel Corp.
Chandler, Ariz.

However arcane fuzzy logic might seem to designers just making its acquaintance, one aspect of fuzzy design remains relatively straightforward: Choosing a microcontroller for a fuzzy system is no more wrenching than choosing one for conventional systems. If care is taken in the selection process, microcontrollers executing fuzzy software can easily meet the performance requirements of fuzzy applications.

Peripheral functionality has not changed for fuzzy logic, but new architectural requirements involve register depth and logic functionality. The number of bits is determined by the resolution required.

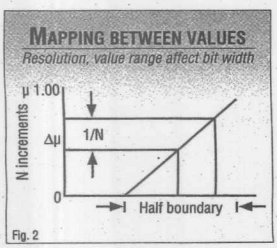


Fig. 2

The use of fuzzy logic opens the system performance envelope in two dimensions: machine IQ (application intelligence) and architecture performance. The machine IQ is a function of the microprocessor/controller performance. Conversion from traditional logic to fuzzy logic permits the designer either to retain the same processor performance and increase the machine IQ, or keep the same machine IQ and reduce the processor performance requirements (see Fig. 1).

This stretched performance is the result of using rules rather than algorithms to control the system. Where many traditional applications incorporate high-order-differential equations to solve control problems, fuzzy logic instead uses a series of rules, significantly reducing performance requirements.

This excess performance can either be eliminated to reduce the system cost or used to create a smarter system. Therefore, some applications can reduce their architectural requirements by moving from a 32-bit processor to 16 bits, or from 16 bits to 8.

Given this freedom of choice and the unfamiliarity of performance expectations of fuzzy logic, the microcontroller possibilities are bewildering. Fortunately, system requirements set the selection criteria, just as they do with conventional systems. These criteria mirror the fuzzy engine and the interface requirements.

Every control application has two elements: an interface to the "real world" for input and output, and the control strategy. The sum of these two elements defines the performance requirements. In both conventional and fuzzy logic systems, the interface is performed by the microcontroller's peripherals. Fuzzy logic implements the control strategy.

Defining the rules by which a fuzzy logic application works is a three-step process: "fuzzification," inference and "defuzzification." Fuzzification maps crisp values to a fuzzy membership, while inferencing evaluates the validity of a term, or linguistic variable, in the rule. Finally, defuzzification translates the fuzzy output to a crisp output. This crisp output enters the real world via the peripherals.

The resolution of the data to be fuzzified and its membership define the bit requirement of the microcontroller. Fuzzy logic is a mapping between crisp values, such as 78x, and fuzzy values, such as the linguistic variable "warm." This mapping transfers crisp values into the membership set or function. The membership value ranges from 0 to 1, with all increments in between valid. Most applications can tolerate a membership resolution of .01. As a result, the membership set can be represented by a byte or 8 bits.

Next, the slope of a term such as "warm" exerts its influence on the resolution. The slope shown in Fig. 2 is 1/2 × Boundary, where Boundary indicates the range of values (such as 65° to 75°F) within which the term applies. The resolution of the crisp value is the Boundary value divided by twice the membership resolution. Application resolution is the range of the universe of discourse (the total range of values for all terms) times the Boundary divided by twice the resolution of the membership.

The range of the crisp values in the universe of discourse and the range of the terms affect the bit requirements. A wide range of input values and a broad term will require more resolution, which results in a wider word requirement. In data control applications, the size of the data file to be evaluated also affects the bit-width requirements. Systems dealing with data analysis and pattern recognition have large data sets and require a wide bit width for addressability.

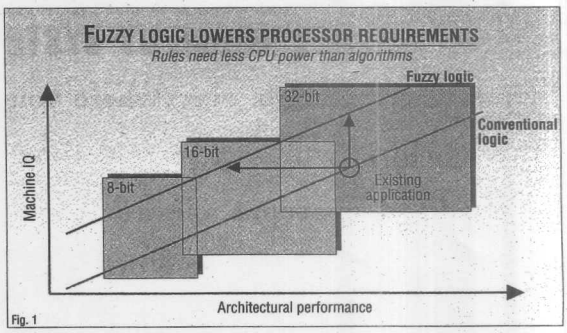


Fig. 1

Inference rules

The set of rules replaces the control algorithm or equation used in conventional systems. Each rule in the set defines an action to be taken if evaluation conditions are met. To adequately define the system, a number of rules must be evaluated each time new input data is acquired or a new output must be generated.

Microcontrollers evaluate the rules in a serial fashion. As a consequence, increasing the number of rules increases the time needed to evaluate them. Fuzzy systems are nondeterministic, and not every rule will participate each time. Worst-case timing is calculated by assuming that every rule does participate each time.

Each time a new input sample is captured, it must be evaluated to generate an output. The time from capturing an input to generating an output is the system loop time. Rule-evaluation time is bounded by the system loop time. This in turn, is limited by the Sampling Theorem, which requires sampling frequency to be at least twice the highest frequency of the sampled signal. The highest frequencies of the input and the output signals define the

Continued on page 54

Artificial intelligence can trace bugs

By John Sambrook
Senior Software Engineer
Applied Microsystems Corp.
Redmond, Wash.

One technique that is sure to find wider usage in embedded-systems development tools is the application of artificial intelligence to allow the tools to make inferences about processor behavior, without the engineer's intervention.

Applied Microsystems Corp. has used such techniques to provide accurate trace disassembly, a chore that is, at best, problematic with new processors that rely on prefetching to improve system performance. In prefetching, the processor accesses instructions from memory before they are required for execution.

However, many of the instructions the microprocessor fetches will never be executed, which makes accurate trace disassembly difficult. That's because trace disassembly takes information from the execution trace; it depends upon viewing a history of the instructions that the microprocessor read from memory.

New disassemblers draw on AI research into knowledge representation.

In prefetching, the processor's bus controller mechanism is continually issuing bus cycles to read instructions into memory before they are required by the execution unit. For the embedded systems designer who is trying to use a raw execution trace to debug a system, this adds a tedious burden. Now, the engineer must spend time deciding which bus cycles correspond to instructions that were actually executed, as opposed to those that were simply fetched but never executed.

This problem is further complicated by such features as store buffers. A store buffer is a special register that holds data waiting to be written to memory. In some microprocessors, data can remain in the store buffer for dozens of bus cycles. When it does, though, analysis of the execution trace is more difficult, because the contents of the store buffer affect the number and ordering of the bus cycles the microprocessor runs.

In interpreting a raw execution trace, designers also have the task of matching data cycles with the instructions that caused them. Prefetching and store buffers greatly complicate this problem, too. In prefetching, the fetched-but-not-executed instructions frequently act like red herrings—they are not visually distinct from fetched-and-actually-executed instructions. Store buffers, depending on their structure and contents, change the order and number of bus cycles run. These two factors alone greatly complicate the task of understanding a raw execution trace.

Recent advances in trace-disassembly technology can be applied to these problems. New disassemblers draw on artificial intelligence research in the areas of knowledge representation, search and belief systems.

Complex translations

A trace disassembler is a software tool—part of an emulator—that translates the contents of a trace buffer into the instructions that were executed by the microprocessor in a form meaningful to hardware and software developers. For early microprocessors, the process involved a reasonably straightforward translation of the bus cycles into executed instructions and data accesses. As microprocessors have become more complex, translation has become increasingly more difficult to perform with accuracy and efficiency.

But when the newest trace disassemblers are asked to disas-

Continued on page 54

Embedded Systems: Part 4

Rules are clear for fuzzy controllers

Continued from page 52
limit of execution time.

For example, if the input is 1 kHz, the sampling frequency must be a minimum of 2 kHz. That means the microcontroller must "go through the loop" every 1/(2 kHz), or 500 μs. Not all of the 500 μs can be allotted to the inferencing process.

Performance determinants

During this time, the microcontroller must capture the input data, perform the inferencing and generate the output. The performance requirements are bounded by the number of rules to be evaluated and the sampling period on the input and output. Elements that affect this performance are fast and autonomous peripherals, adequate internal data storage and efficient architecture for logical operations. Less time spent on servicing input and output provides more time to inference.

CPU execution gates the performance. One architectural feature that improves performance is a register file, which permits storing the term points and the membership of terms in the registers. As a result, performance is enhanced by eliminating external memory cycles.

Usually, no more than nine terms are used, and each has four points per term. Therefore, the amount of internal RAM required is 36 bytes or less per term. Seven is a more typical number of terms, requiring only 28 bytes. A four-input system would then require 112 bytes to map all terms.

Defuzzifying the output

Defuzzification is performed by calculating the area under curves of the output terms. Multiply and accumulate functions are used to calculate the crisp output value, and 16-bit mathematics will enhance performance. For large rule sets or minimum loop time, 16-bit microcontrollers should be considered.

Software implements fuzzy logic with standard microcontroller instructions. The CPU does the fuzzification, inference and defuzzification via software while the peripherals capture the input data and provide an output. A single-chip microcontroller such as the Intel MCS-96 can perform both functions—interface and control—using software. The advantages of using a standard microcontroller are cost, availability, a well-known standard architecture and programming tools that are popular and easy to use.

NEW DEVELOPMENT SOFTWARE AIDS IN DESIGN, DEBUG

Putting fuzzy to work

Continued from page 46
degrees of truth. We then decide on a single action, which is influenced by all of these "suggestions."

Older fuzzy algorithms merely selected the largest fuzzy output (max. defuzzification), but that technique does not consider the contribution of other rules that could be almost as significant. A weighted average of singleton fuzzy outputs is a better method for common control applications.

Picking applications

The first step in applying fuzzy logic to your application is to examine it to determine what portion can benefit from a fuzzy approach. A decision that is based on subjective-sounding inputs, or a complex non-linear relationship that is not readily expressible as a mathematical formula, is a good candidate. Many parts of a typical control problem, such as switch inputs and displays, will still be solved with conventional (non-fuzzy) programming techniques.

The fuzzy problem will break down into two somewhat independent parts. One involves capturing and expressing the knowledge of how the system works (developing the knowledge base). The other involves writing a software program (the fuzzy inference engine) that can process the information in the knowledge base to transform real-time system inputs into the appropriate control action.

The application expert can develop the knowledge base without any significant knowledge of embedded controller programming. The fuzzy inference engine is developed by an embedded system programmer using conventional tools and techniques. The programmer will also be involved with the fuzzy development tool that translates rules and membership functions into ROM data structures because the knowledge base data must be processed by the inference program.

Though largely independent, these two people or groups must come together to discuss each other's requirements. Together they must decide how often the inference engine must sample inputs and how quickly the engine must produce an output in response to those inputs. From those discussions, the programmer can decide on specific strategies for each of the three parts of the inference engine.

> The first step in applying fuzzy logic is to examine your application.

Another important step is to determine the inputs and outputs, giving each a linguistic name. At that point, you can try writing a few simple rules that describe what the control system is intended to do.

From those trial rules, the input and output parameters should become apparent. Think of what other conditions could influence the results and include them as inputs as well. Inputs can include time or derived quantities such as the difference between current temperature and setpoint or rate-of-change of temperature. Conventional techniques are used to precalculate such values so they can be presented to the fuzzy inference program.

Next, determine minimum and maximum values for each input and output and split those into ranges of values that can be described by linguistic labels. There will be one membership function for each linguistic label of an input or output. A membership function is a graphical representation of what each linguistic label "means." Membership functions for adjacent input labels almost always overlap. Singletons are often used for output membership functions because they represent a specific control setting and require less math to defuzzify.

Commercial fuzzy development tools from Aptronix, Togai Infralogic, Inform and others provide a means of entering rules and membership functions as well as assisting in the debug process. The tools emulate what the inference engine would do for all combinations of inputs, and display the results in three-dimensional control surfaces that relate two inputs to one output at a time. You can identify a spot on a control surface and quickly see which rules and membership functions have an influence. Membership functions are entered graphically.

The commercial tools generate the data structures and inference code to process the knowledge base. The embedded system programmer can then incorporate the fuzzy block into the larger application system.

AI: smart path to trace disassembly

Continued from page 52
semble a region of the trace buffer, they treat the request as a search problem and use a depth-first search algorithm to look for a trace-disassembly solution that is consistent with the data in the trace buffer.

The disassembler does that by creating an elaborate model of the target processor. The disassembler starts with an empty model in which it marks all target processor resource values (e.g., registers) as "unknown."

As the trace disassembler goes into action, it reads information from the execution trace, looking for instruction-fetch bus cycles. The set of "acceptable" instruction-fetch bus cycles is constrained by three knowledge sources: the knowledge built into the disassembler, the knowledge accumulated during this disassembly and the knowledge accumulated in previous disassemblies, if any, of this particular execution trace. By enforcing the constraints generated from the knowledge sources, the disassembler is able to prune the search space to one that can be searched in a reasonable amount of time, currently on the order of 1/2 to 3 seconds.

Tracking data cycles

For instructions that generate data bus cycles, the disassembler must find the data cycles that were caused by the execution of the instruction. Again, the disassembler relies on its various knowledge bases to make that process efficient. As data cycles are paired with the instructions believed to have caused them, the disassembler updates its set of beliefs and continues processing.

Execution traces are rich with uncertainties that have to be resolved. Depth-first search (with the associated backtracking) is the general method the disassembler uses to guide its search for a consistent solution. Again, the various knowledge sources maintained by the disassembler can make the process both extremely accurate and reasonably efficient.

A useful side effect of this process is that the disassembler computes elaborate information on the state of the target microprocessor after each instruction execution. For example, it can keep track of microprocessor register values and display them. Thus, the user can generally see not only the bus-cycle data corresponding to a particular instruction execution, but also the corresponding register data. That also applies to instructions with no associated bus cycles, e.g., register-to-register moves. The disassembler is generally able to provide the actual data values that were moved.

> The disassembler creates a model of the target processor.

It is interesting to note that the disassembler infers displayed register values based on the observed behavior of the microprocessor, since those are not directly observable off-chip. The ability to see register values as they were at the time of execution greatly increases the designer's understanding of the behavior of the system.

The technology is intelligent in that its elaborate model helps the disassembler recognize its own mistakes. As a result, the programmer can perform disassembly faster and more accurately.

Intelligent trace disassemblers are just beginning to become available, and they promise to be even more useful in the future. For example, such a tool might function as a virtual database that contains a large amount of extremely detailed information on the last microprocessor run. That database can be used as input to other emulator subsystems, such as performance analysis, real-time operating systems support and high-level debuggers.